LEGOS:un langage modulaire (a modular programming language)

J.L. Bouchenez, M. Loyer, F. Prusker, J.C. Sogno

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: 954 90 20

Rapports Techniques

N° 1

LEGOS:
UN LANGAGE MODULAIRE

LEGOS:
A MODULAR PROGRAMMING LANGUAGE

Jean-Louis BOUCHENEZ
Michel LOYER
Francis PRUSKER
Jean-Claude SOGNO

Avril 1981

LEGOS : UN LANGAGE MODULAIRE

LEGOS : A MODULAR PROGRAMMING LANGUAGE

J.L. Bouchenez, M. Loyer, F. Prusker, J.C. Sogno

INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78150 - LE CHESNAY

LEGOS : UN LANGAGE MODULAIRE

Résumé :

Le langage LEGOS est un langage de programmation orienté vers l'écriture de logiciels de base, comportant des mécanismes de programmation modulaire simples mais puissants.

Ce rapport comprend une présentation de ces mécanismes et une description du langage proprement dit.

LEGOS : A MODULAR PROGRAMMING LANGUAGE

Abstract :

The language LEGOS is a programming language designed for system implementation, with simple but powerful facilities for modularization.

This report includes the presentation of these facilities and a description of the language itself.

SOMMAIRE

PRESENTATION DU LANGAGE LEGOS

Caractéristiques générales

Le langage LEGOS est un langage de haut niveau orienté vers l'écriture de logiciels de base -c'est-à-dire, d'applications utilisant et/ou comportant un noyau-système. L'objectif de LEGOS est de permettre d'écrire, dans un unique langage, une application et sa partie système, voire un noyau-système sur machine nue.

Le langage LEGOS est dérivé de PASCAL auquel il emprunte sa structure déclarative et la plupart de ses instructions. Certaines notions ont été supprimées comme le type real, d'autres ajoutées comme le type word (qui permet de manipuler le mot-machine). Certains autres traits sont inspirés de langages comme LIS, MESA, MODULA, MODULA-2.

LEGOS offre également la possibilité de structurer logiquement une application grâce à la notion de module. Il permet également de décrire des activités s'exécutant en parallèle. LEGOS se veut un langage indépendant machine ; cependant l'interface avec un éventuel environnement externe (librairie, système..) peut être décrit dans le langage.

Les mécanismes de programmation modulaire dans LEGOS

Le langage LEGOS permet de décrire une application sous forme d'un ensemble d'unités élémentaires appelées modules. (En général, un module regroupe tous les objets propres à une entité logique). Un mécanisme simple de liaison entre les modules permet de refléter, dans le langage, les liens logiques existant entre ces diverses unités.

De plus, cette structuration permet de mettre en oeuvre un mécanisme de compilation séparée des modules, garantissant la cohérence globale entre les différentes parties ainsi compilées.

Ce choix consistant à calquer la structuration de compilation séparée sur la structuration logique, permet de ne manipuler qu'un seul objet, le module, à la fois concret comme unité de compilation et abstrait comme unité logique de l'application.

Les mécanismes d'exécution dans LEGOS

Le langage LEGOS permet de décrire des activités susceptibles de s'exécuter concurremment grâce à des modèles de tâche dont des exemplaires peuvent être créés à l'exécution. Chaque modèle de tâche est lui-même structuré comme un programme séquentiel classique.

Le langage LEGOS ne présuppose aucune stratégie quant à la synchronisation des tâches et n'offre donc pas d'instructions particulières d'activation-désactivation. L'activation et la désactivation se feront à l'aide de procédures appartenant à un noyau-système (éventuellement écrit en LEGOS) standard ou propre à l'utilisateur.

Le choix de la stratégie de synchronisation étant fait au niveau de ce noyau-système et non au niveau du langage, cela offre une plus grande souplesse pour tenir compte du contexte (type du matériel, type de l'application,..). Cela évite également de devoir obligatoirement passer par un exécutif unique comme c'est le cas lorsque le choix de stratégie a été fait au niveau du langage.

Pour faciliter l'écriture et la mise au point de programmes écrits en LEGOS, nous réalisons un environnement comprenant un bibliothécaire de gestion des programmes, un éditeur syntaxique et un paragrapheur.
Cet environnement fera l'objet d'un rapport ultérieur.

CHAPITRE 1

LA MODULARITE DANS LE LANGAGE LEGOS

1 - LES MECANISMES DE PROGRAMMATION MODULAIRE

1.1 - Structure générale d'une application LEGOS

Une application LEGOS se présente sous la forme d'un ou plusieurs modules, compilables séparément.

Chaque module consiste en :

- une série de déclarations d'objets (constantes, types, variables, en-têtes et corps de sous-programmes), formant un ensemble logique cohérent pour la compilation ;
- et éventuellement, une liste d'instructions.

Ces objets doivent être de niveau global : il n'y a pas d'imbrication de modules, ni de modules dans les corps de sous-programmes.

Exemple

```
module m ;
   const nmin = 0 ;
         nmax = 10 ;
type t = array [nmin...nmax] of integer ;
   var t1, t2 : t ;
       flag : boolean ;
   action p = procedure  (v : t) ;
   body p ;
        begin
        ...
        end p ;
begin
flag := false
end m ;
```

Un module m peut nommer dans une clause use d'autres modules. Ces modules constituent le contexte de compilation de m. Dans m, il est alors possible de référencer tous les objets déclarés dans les modules du contexte.

Exemple :

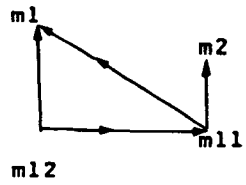
```
module m1 ;
...
end m1 ;

module m2 ;
...
end m2 ;

module m11 use m1, m2 ;
...
(*les objets de m1 et m2 sont accessibles*)
...
end m11 ;

module m12 use m1, m11 ;
...
(*les objets de m1 et m11 sont accessibles*)
...
end m12
```

Schéma des liens entre ces modules :



Notons que la liaison use n'est pas transitive ; ainsi, dans l'exemple, il n'est pas possible dans m12 d'avoir accès aux objets déclarés dans m2.

Un module ne peut être compilé que si son contexte peut être construit ; c'est-à-dire, si les modules, nommés dans la clause use, ont déjà été compilés.

Dans l'exemple ci-dessus, les ordres possibles de compilation sont :

m1, m2, m11, m12

et

m2, m1, m11, m12

1.2 - Développement modulaire des procédures

En LEGOS, nous avons séparé en deux entités la notion habituelle de procédure.

L'une, introduite par action, correspond à l'en-tête de la procédure (partie spécification) ; l'autre, introduite par body, correspond au corps de la procédure (partie réalisation).

L'en-tête et le corps associé peuvent alors apparaître dans les modules différents. Le module, contenant le corps associé à une déclaration de procédure, doit mentionner le module contenant cette déclaration dans son contexte.

On peut ainsi facilement développer plusieurs réalisations pour une même spécification.

Exemple

```
module fichier ;
  type file = ... ;
  action open = procedure(inout:file) ;
  action close = procedure(inout:file) ;
end
```

```
module realis1 use fichier ;
        ...
    body open ; ... ;
    body close ; ... ;
        ...
 end realis1
```

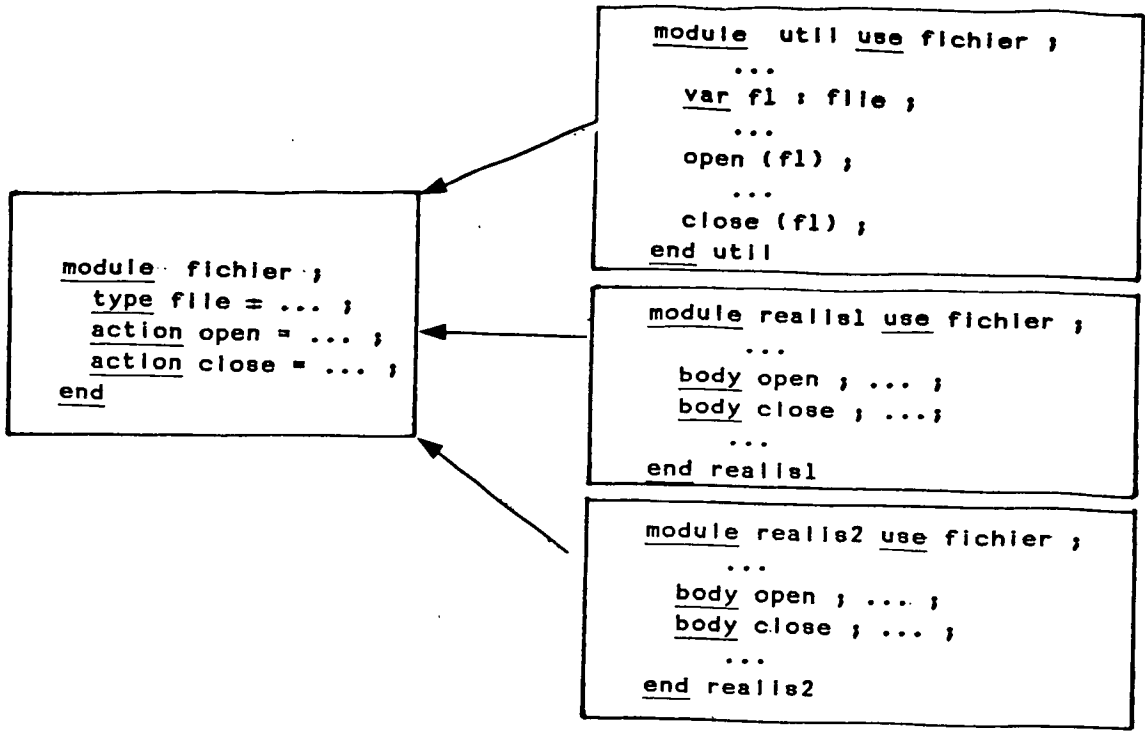
```
module realis2 use fichier ;
        ...
    body open ; ... ;
    body close ; ... ;
        ...
 end realis2
```

Le choix entre ces différentes réalisations possibles d'un sous-programme se fera au moment de la constitution d'un programme lors d'une phase d'édition de liens.

1.3 - Programme élémentaire

Un programme LEGOS élémentaire est constitué d'un ensemble de modules ayant des liens de visibilité entre eux.

Reprenons l'exemple décrit précédemment en lui adjoignant un module utilisateur :



A partir de ces modules, on peut par exemple construire les deux programmes suivants :

```
program P1 = (fichier, realis1, util)
program P2 = (fichier, realis2, util)
```

Lors de la constitution du programme, il est vérifié que l'ensemble de ces modules est cohérent ; c'est-à-dire, pour tout module m appartenant à la liste des modules d'un programme p, on doit avoir :

- si m mentionne dans sa clause use un module n, alors n appartient à la liste des modules de p ;
- si m contient la déclaration d'un sous-programme r, sans que le corps associé apparaisse dans m, alors il existe, dans la liste de p, un unique module n, référençant m dans sa clause use et contenant le corps associé à r.

1.4 - La notion d'interface

Le découpage d'une application en modules tel que nous l'avons vu jusqu'ici permet de limiter, en nombre, les liens de visibilité entre les objets, mais il ne permet pas d'affiner la teneur de ces liens. Ainsi, on ne peut exporter un type en cachant sa structure ou exporter une variable en lecture seulement.

Pour ce faire, a été introduit en LEGOS un outil de structuration complémentaire : l'interface.

Une interface permet d'encapsuler un ou plusieurs modules, en ne laissant visibles de l'extérieur que certains objets. Ces objets peuvent être des constantes, des types, des variables et des en-têtes de sous-programmes. L'interface est construite a posteriori à partir de modules déjà compilés.

Les types ainsi exportés peuvent l'être soit avec leur structure visible (elle doit alors être répétée dans l'interface) soit avec leur structure cachée (attribut hidden).

Les variables exportées peuvent l'être en étant protégées contre toute modification et ne peuvent alors qu'être consultées (attribut protected).

Il est aussi possible de renommer les objets exportés.

Exemple

Reprenons l'exemple du module fichier décrit précédemment. Il est souhaitable de cacher les détails de structure du type file de façon à ce que des variables de ce type ne soient manipulées que par les procédures open et close. Ceci peut se faire au moyen de l'interface suivante (qui, de plus, renomme les objets exportés) :

```
module fichspec interface fichier ;
   type fich is file = hidden ;
   action ouvrir is open = procedure (inout f : fich) ;
          fermer is close = procedure (inout f : fich) ;
end fichspec
```

Ces interfaces peuvent être utilisées dans la compilation d'un module qui les mentionnent dans une clause use. Les objets, décrits dans les interfaces, peuvent alors être utilisés dans ce module en respectant les contraintes nouvelles qui ont pu leur être adjointes (hidden, protected).

```
Exemple :
module util use fichspec ;
       ...
   var fl : fich ;
       ...
       ouvrir (fl) ;
       ...
       fermer (fl) ;
end util
```

Un module m peut mentionner conjointement dans sa clause use des modules et des interfaces. Cependant dans tous les cas, il ne doit y avoir qu'un mode de visibilité pour un module importé soit directement, soit via une interface.

Une interface LEGOS consiste simplement en un calcul sur les environnements que constituent les modules encapsulés, de façon à vérifier la cohérence de l'interface à produire un environnement restreint. Il n'y a pas de code objet associé à une interface.

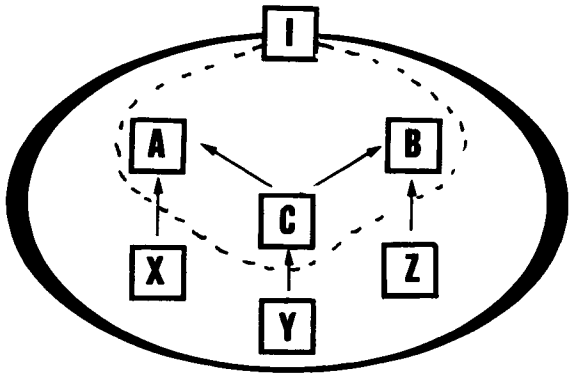
Les seuls modules qu'il est nécessaire d'encapsuler dans une interface sont ceux utiles à la compilation de l'interface. Ainsi, dans l'exemple précédent, seul le module fichier a été encapsulé ; il n'était pas nécessaire de lui adjoindre le module contenant les corps des sous-programmes.

L'interface est aussi le mécanisme qui permet d'utiliser dans une application LEGOS des objets non-écrits en LEGOS. Ces interfaces particulières servent alors à donner une description en LEGOS de ces objets.
Ceci permet en particulier de pouvoir utiliser dans une application les fonctions d'un noyau-système, les procédures d'entrée/sortie ou les mécanismes de synchronisation entre tâches déjà existants.

1.5 - Programme général

1.5.1 - Association d'un programme et d'une interface

On a vu que seuls les modules nécessaires à sa compilation sont référencés dans la clause use d'une interface. Si certains de ces modules contiennent des en-têtes de sous-programmes, les corps associés peuvent apparaitre dans des modules non spécifiés dans l'interface. On a alors une structure de la forme suivante :



où I est une interface sur A, B, C
et X, Y, Z réalisent les objets spécifiés respectivement dans A, B, C.

Un programme est associé à une interface par l'une des deux commandes suivantes :

a) program P = (A, B, C, X, Y, Z) defines I
b) program P defines I (si P a déjà été construit par une commande programme antérieure).

Si P n'est pas déjà construit (cas a) on construit P comme expliqué précédemment et on vérifie que P constitue bien une réalisation de l'interface I . Pour cela il faut que les modules apparaissant dans la clause use de I appartiennent à P.

On remarquera qu'il est possible d'associer un même programme à plusieurs interfaces différentes ; chacune d'elles constitue un angle de vision différent du programme.

Exemple :

Si J est une interface sur A, B on peut avoir les commandes suivantes :

```
program P = (A, B, C, X, Y, Z)
program P defines I
program P defines J
```

On peut également associer plusieurs programmes à une même interface ; chaque programme constitue une réalisation particulière de la spécification de L'interface.

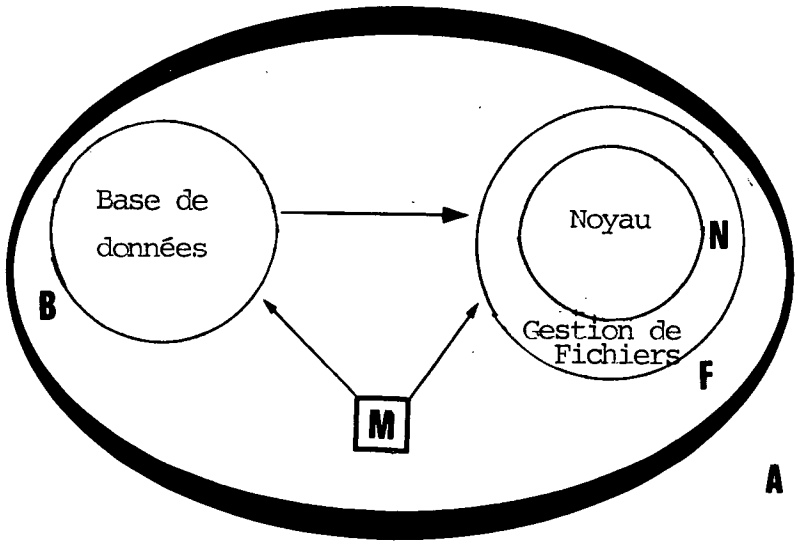
Exemple :

Si les modules X', Y' et Z' correspondent à une autre réalisation de A, B, C on peut avoir pour l'interface I les deux programmes suivants :

```
program P = (A, B, C, X, Y, Z) defines I
program P' = (A, B, C, X', Y', Z') defines I
```

1.5.2 - Programme général

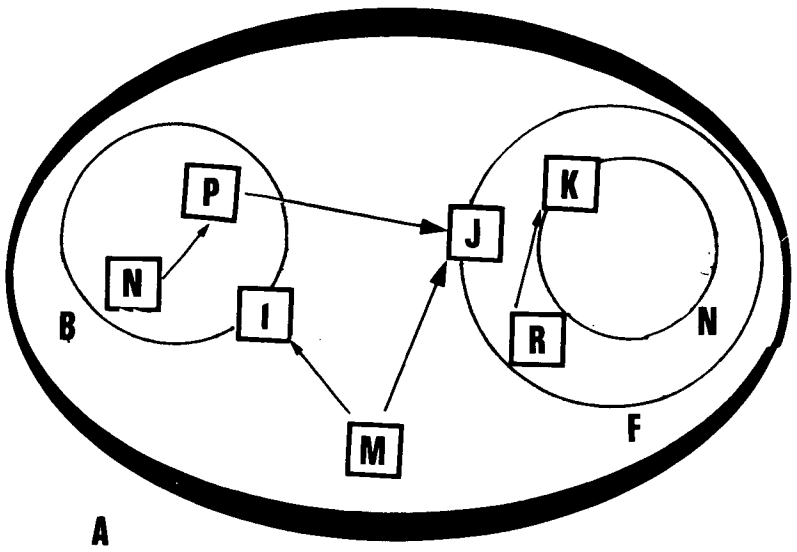
Jusqu'ici, nous n'avons vu que des programmes élémentaires ne comportant que des modules. Un programme LEGOS général est en fait constitué d'un ensemble de modules et de programmes ayant des liens entre eux. Considérons l'application suivante :



Cette application A comprend un module M qui utilise une base de donnée B et un système de gestion de fichier F, qui utilise lui-même les services d'un noyau de système. En LEGOS, cette application aurait la structure suivante :

- A, B, F et N sont des programmes constitués de modules et de programmes ;
- A est un programme "exécutable" ;
- B, F et N sont des programmes librairies dont les services sont utilisés par d'autres programmes via une interface associée.

Cette structure peut être traduite par le schéma suivant où sont précisés les modules constituants et les modules interfaces associés aux programmes A, B et F.



Nous voyons par exemple que le programme B comprend deux modules, N et P, et est accessible via l'interface I. Le programme J comprend le module R et le programme N ; il est accessible via l'interface J.

Nous indiquons ci-dessous la liste des commandes nécessaires pour constituer cette application :

program N = (...) defines K

```
program F = (R, K = N) defines J
program B = (N, P) defines I
program A = (M, I = B, J = F)
```

Pour chaque programme, il faut préciser l'interface à travers laquelle ses services sont utilisés ; par exemple, dans la commande de définition du programme A, on indique que le programme B est accessible via l'interface I. Rappelons en effet que plusieurs interfaces peuvent être associées à un même programme.

1.6 - Le bibliothecaire

1.6.1 - Présentation

Les mécanismes de compilation séparée qui ont été choisis :

- compilation de modules avec contextes,
- utilisation de librairies,

conduisent à une multiplication des objets à gérer par l'utilisateur. Pour faciliter sa tâche, nous avons conçu un outil, le bibliothécaire qui a pour fonctions de :

- gérer les différents objets associés aux compilations des modules, aux constructions et aux exécutions des programmes (binaires, contextes, interfaces...) ;
- vérifier la cohérence des commandes ;
- renseigner l'utilisateur sur l'existence et le contenu des différents objets qu'il gère.

Dans toutes les commandes qu'il a sa disposition, l'utilisateur nomme toujours des modules et des programmes ; le bibliothécaire se charge en fonction de la commande et de ces noms de fournir les objets correspondants nécessaires à la réalisation de la commande.

1.6.2 - Compilation d'un module

La compilation d'un module est lancée par la commande :

compile nom de fichier.

Le fichier nommé contient le texte source du module (ou des modules) à compiler.

La compilation de l'en-tête d'un module provoque l'appel au bibliothécaire qui vérifie la cohérence du contexte constitué par les modules cités dans la partie use.

Exemple : module M use X,Y,Z ;

En cas d'impossibilité de créer le contexte de M (c'est-à-dire si X, Y ou Z n'a pas été compilé), un message d'erreur est édité et la compilation est interrompue.

1.6.3 - Construction d'un programme

A partir de binaires et de programmes déjà construits, on peut constituer un nouveau programme à l'aide d'une commande de la forme :

```
program progident = (listident) ; [defines interident]
```

avec

progident : nom du programme
listident : liste de modules et de programmes constituant le programme
interident : nom de l'éventuel interface associé au programme.

Les noms de programme définis par les commandes program sont gérés par le bibliothécaire. Ils sont utilisés

- pour l'exécution d'un programme dans une commande:
  execute nom de programme.

- pour la construction d'autres programmes ; dans ce cas ils peuvent être considérés comme des librairies. Il n'y a pas de commande spéciale de mise en librairie ; tout programme peut servir de librairie à condition qu'il ait été construit avec une interface.

1.7 - Conclusion

Les mécanismes de compilation séparée introduits dans LEGOS s'efforcent essentiellement de garantir la cohérence d'une application développée de façon modulaire, tout en laissant au programmeur autant que faire ce peut, le choix de la méthodologie et du découpage de son application.

De plus, ces mécanismes restent suffisamment simples pour n'avoir que des répercussions limitées et localisées sur la structure du langage LEGOS et sont par conséquent immédiatement applicables à d'autres langages de la famille PASCAL.

2 - LES MECANISMES D'EXECUTION

2.1 - Généralités

Un programme LEGOS, à l'exécution, se compose d'une ou plusieurs tâches qui s'éxécutent concuremment.

Chacune de ces tâches possède ses objets propres qui ne sont pas accessibles aux autres tâches.

Il existe également des objets globaux (ou communs), accessibles à toutes les tâches. Ces objets peuvent être des données, des sous-programmes...

Les objets globaux sont élaborés au début de l'éxécution d'un programme et existent jusqu'à la terminaison du programme.

Les tâches (et par conséquent, leurs objets propres) sont elles créées dynamiquement au cours de l'éxécution du programme à partir d'un modèle de tâche.

2.2 - Création des tâches

La création d'une tâche se fait à l'aide de l'instruction create qui a la forme suivante :

```
create t from mod(p1,p2,p3) ;
```

où t est une variable de type prédéfini repère de tâche (ref),
mod est un modèle de tâche
et p1,p2,p3 sont les valeurs des paramètres d'initialisation de la tâche.

Cette instruction crée une nouvelle tâche T qui est repérée par t, mais il n'y a pas activation de T.

Le lien entre la tâche T et le repère t est analogue à celui existant en Pascal entre un pointeur et un objet alloué dynamiquement.

Ainsi, on peut comparer deux repères, les affecter l'un à l'autre. La constante prédéfinie notask permet d'indiquer qu'aucune tâche n'est associée à un repère.

Un point important est qu'il n'y a aucun lien entre la durée de vie d'une tâche et celle de son repère.

Un modèle de tâche est obligatoirement déclaré au niveau le plus externe d'un programme sous une forme assez semblable à celle des procédures (cf Chapitre 2 4.5).

Il n'y a donc par imbrication de modèles, ce qui permet d'assurer qu'une tâche ne référence que ces objets propres ou des objets globaux.

Cependant, durant son déroulement, une tâche T peut créer une autre tâche S. Mais cela ne crée aucun lien prévilégié entre T et S. En particulier, cela ne crée aucune relation entre la durée de vie de S et celle de T.

2.3 - Activation des tâches

Volontairement, le langage LEGOS ne fournit aucune instruction pour effectuer l'activation, la désactivation et les synchronisations des tâches. Seule cette attitude nous parait raisonnable pour un langage d'écriture-système, car l'introduction dans un langage de mécanismes d'activation-désactivation, aussi simples soient-ils, implique toujours un choix de stratégie et donc l'existence d'un exécutif minimal lié au langage.

Ces mécanismes (et donc ces choix de stratégie) pourront être décrits en LEGOS par l'utilisateur à l'aide d'interfaces (cf Chapitre 1 1.4).

Ils pourront aller, selon les besoins, depuis le simple changement de contexte jusqu'aux algorithmes d'ordonnancement les plus sophistiqués.

Un mécanisme minimal sera prédéfini pour toute implémentation de LEGOS et reflétera les caractéristiques propres à la machine correspondant à cette implémentation.

CHAPITRE 2

LE LANGAGE LEGOS

1 - NOTATION POUR LA DESCRIPTION SYNTAXIQUE

La description syntaxique est faite selon un formalisme proche de celui de Backus-Naur. La grammaire se présente comme un ensemble de règles de la forme <S> = E où <S> désigne un non-terminal de nom S et où E est une expression indiquant l'ensemble des phrases dérivant de S.

Un non-terminal est désigné par un nom reflétant sa signification ; ce nom est noté entre les symboles < et >.
Exemple : <bloc>, <procédure>.

Les terminaux sont

- soit des unités syntaxiques propres au langage comme les mots-clés (notation : begin pour begin, end pour end) ou des symboles réservés (notation : ':=' pour :=, '+' pour +).

- soit des représentants d'une classe lexicographique (notation : ident pour tout identificateur, chaîne pour toute chaîne de caractères..).

- Le symbole | sépare les alternatives d'une règle.
- Le symbole . termine une règle.
- Le symbole ϕ dénote l'alternative vide.
- [E] dénote E ou ϕ.
- {E} dénote ϕ ou E ou EE ou EEE...
- Les parenthèses permettent de grouper termes et facteurs.

2 - VOCABULAIRE

Les phrases du langage sont construites à partir d'un vocabulaire constitué de mots-clés, de symboles réservés, d'éléments appartenant à des classes lexicographiques (identificateurs, nombres décimaux...) et de commentaires.

2.1 - Identificateur

Un identificateur est une séquence de lettres, de chiffres ou de blancs soulignés commençant par une lettre.
Exemple : X AN1980 JOURS_DE_LA_SEMAINE.
Notation syntaxique : ident.

2.2 - Nombre décimal

Un nombre décimal est constitué d'une suite de chiffres.
Exemples : 0 10 349.
Notation syntaxique : int.

2.3 - Nombre hexadécimal

Un nombre hexadécimal est représenté par un dièse suivi par une séquence comprenant des chiffres ou les lettres A,B,C,D,E,F.
(A = 10,...,F = 15).
Exemple : #1A #1BBC #A1B9.
Notation syntaxique : hexa.

2.4 - Caractère

Un caractère est représenté soit par un caractère entre guillemets, soit par un dièse suivi entre guillemets d'un nombre hexadécimal.
Exemple : "A" "+" (""" désigne la caractère ")
#"C1" #"F5".

Notation syntaxique : char.

Remarques : - Il n'y a pas un ensemble propre de caractères pour le langage. Cet ensemble et son codage dépendent de l'implémentation.
- La seconde notation permet d'indiquer directement le codage du caractère. Ceci permet d'utiliser les caractères non-standard (BELL, ETX, EBL..).

2.5 - Chaîne

Une chaîne est représentée par une suite de caractères quelconques compris entre deux apostrophes ('). Si ces caractères sont représentés sous forme hexadécimale, la chaine est précédée d'un dièse (#).
Exemple : 'ABCD' #'C1C2C3C4'.
Notation syntaxique : chaîne.

Remarques : - Pour introduire le symbole ' dans une chaîne il suffit de le doubler ; ainsi 'AB''CD' représente la chaîne AB'CD.
- Une chaîne peut être scindée en plusieurs parties reliées par le symbole &. Ces parties pouvant être séparées par des blancs ou des fins de ligne.
Exemple : 'ABC' & 'DEF' est équivalent à'ABCDEF'.
'AB' & 'CD' & 'EF' est équivalent à'ABCDEF'.
#'C1C2' & 'C3' est équivalent à #'C1C2C3'.
& n'est pas un opérateur entre deux chaînes mais simplement une facilité d'écriture (comme le trait d'union entre deux syllabes d'un mot).

2.6 - mots-réservés

access action and andw array
begin body boolean
case char const create
down
else end endcase endif endloop eor eorw
for from function
goto
hidden
if in inout integer interface is
label loop
mod module
not notw
of or orw others out
pointer procedure protected
range record ref
sar scl scr sll slr
task then to type
unspecified until up use
value var
when while word

2.7 - Commentaire

Un commentaire est une suite de caractères d'une même ligne compris entre (* et *).
Exemple : (* CECI EST UN COMMENTAIRE *).

3 - MODULES

L'unité de compilation (et d'écriture) est le module. Une application est constituée d'un ensemble de modules ayant entre eux des liens de visibilité. Ces liens sont expliqués au Chapitre 1.

```
<module> = <en-tête> {<déclaration>}  {<algorithme>} [<inst-part>] end
           [ident] .
<en-tête> = module ident [use<ident-list>] ';'.
<déclaration> = const {<const-dcl>}  | range {<range-dcl>} | type
                {<type-dcl>} | var {<var-dcl>} | action {<action-dcl>} .
<ident-list> = <ident> {',' ident}.
```

Un module est caractérisé par son nom qui suit le mot module. Ce nom peut être répété derrière le end terminant le module.

Dans l'en-tête d'un module, les noms des modules nécessaires à la compilation de ce module sont indiqués dans une liste suivant le mot use. Viennent ensuite les déclarations : constantes, intervalles, types, variables, spécifications d'actions (en-tête de sous-programmes ou de modèles de tâches). Puis on trouve les réalisations d'actions ou algorithmes (corps de sous-programmes). Enfin, un module peut comporter un bloc d'instructions.

4 - DECLARATIONS ET ALGORITHMES

4.1 - Constantes

Une déclaration de constante associe un identificateur à une valeur.

```
<const-dcl> = ident '=' <constante>';'.
<constante> = <ent-exp> | <bool-exp> | <word-exp> | chaîne | int |
              hexa | char | ident.
```

La dernière alternative permet de renommer une constante.

Lorsqu'une constante est définie par une expression, cette expression doit être calculable lors de la compilation (expression de constantes).

Les chaînes mises à part (cf. 5.2), chaque constante possède un type qui peut être integer, boolean, word, char. (Notons que les nombres hexadécimaux sont de type word).

Certaines constantes sont prédéfinies :

- true et false : constantes booléennes ordonnées par la relation false < true.
- minint et maxint : constantes entières définissant respectivement les bornes inférieure et supérieure du sous-ensemble des entiers correspondant à une implémentation donnée.
- nil est une constante permettant d'indiquer qu'un pointeur ne repère aucun objet.
- notask est une constante permettant d'indiquer qu'un repère de tâche ne repère aucune tâche.

```
Exemple : const   vrai = true ;
                  faux = false ;
                  nom  = 'legos' ;
                  quantum = 100 ;
                  dblequantum = 2 * quantum ;
```

4.2 - Intervalles

Une déclaration d'intervalle associe un identificateur à un intervalle d'entiers (bornes incluses).

```
<range-dcl> = ident '=' <intervalle> ';'.
<intervalle> = <ent-exp>'..'<ent-exp> | ident.
```

```
Exemple :  range chiffre = 0..9 ;
                 heure   = 1..24 ;
                 entiercourt = -128..127 ;
```

Remarque : Les deux bornes doivent être calculables à la compilation (expressions de constantes entières).

4.3 - Types

Une définition de type associe un nom à un type de donnée, qui détermine l'ensemble des valeurs possibles des variables de ce type.

<type-dcl> = ident '=' <type> ';'.
<type> = type de base | <tableau> | <enregistrement> | <pointeur>.

4.3.1 - Types de base

Les types de base sont : integer, boolean, char, word, unspecified. et ref.

integer : une variable de type integer peut prendre des valeurs comprises entre minint et maxint.

boolean : une variable de type boolean peut prendre la valeur false ou la valeur true.

char : une variable de type char peut prendre une valeur quelconque du jeu de caractères disponible pour une réalisation donnée. L'ordre des caractères est fonction de leur codage en machine.

word : une variable de type word peut prendre comme valeur une quelconque configuration du mot-machine. Cet ensemble de valeurs dépend donc de la machine.
Ce type peut être considéré comme packed array [0..n] of bit où n+1 est la taille du mot machine et où bit est une redéfinition du type boolean.
Chaque bit du mot machine est assimilé à un booléen qui vaut false si le bit est à 0 et true s'il est à 1.
Une valeur peut être assignée à une variable v de type word soit globalement soit bit à bit (la notation v[i] désigne le ième bit de v).

unspecified : une variable de type unspecified verra son type précisé à l'exécution par l'utilisation de pointeurs. Elle correspond à la réservation d'un mot-machine ; les seules opérations définies sont l'égalité, l'inégalité et l'affectation entre variables de ce type.

ref : une variable de ce type permet de repérer une tâche crée à l'exécution par l'instruction create. Les seules opérations définies sont l'égalité, l'inégalité et l'affectation entre variables de ce type.

4.3.2 - Types structurés

4.3.2.1 - tableaux

<tableau> = array '[' <intervalle> ']' of (<type> | ident).

Un tableau est une collection d'éléments de même type. Ce type est indiqué par un identificateur de type ou par une définition explicite. Chaque élément du tableau est repéré par un indice entier compris entre la borne inférieure et la borne supérieure précisées par l'intervalle.

```
Exemple : type  t1 = array  [0..25]  of boolean ;
                t2 = array  [1..10]  of array  [2..11]  of integer ;
                t3 = array  [heure]  of word ;
                t4 = array  [-5..10]  of t3 ;
```

4.3.2.2 - Enregistrements

<enregistrement> = record {<champ>} <champ> end.
<champ> = <ident-list>':'(<type> | ident)';'.

Un enregistrement est une collection d'éléments (dits champs) de types quelconques. Chaque champ est repéré par un identificateur. Ces identificateurs permettent de référencer explicitement chaque champ d'une variable de type enregistrement.

```
Exemple : type   t4 = record
                      a : integer;
                      b,c : char;
                      d : record
                          e : boolean;
                          f : array  [0..5]  of word ;
                          end ;
                      end ;
```

4.3.3 - Pointeurs

<pointeur> = pointeur to (ident | type de base).

Les pointeurs permettent un accès indirect à une donnée (cette donnée doit être une variable ou un élément de variable). Un pointeur ne peut référencer qu'un type spécifique de données. Ce type peut être soit un type de base soit le nom d'un type déclaré.

Entre pointeurs de même type sont définies les opérations d'affectation et d'égalité-inégalité. Deux pointeurs sont de même type s'ils pointent des objets de même type.

Soit p1 et p2 deux pointeurs sur le même type d'objet
p1 := p2 signifie faire pointer p1 sur le même objet que p2
p1 = p2 signifie tester si p1 et p2 pointent le même objet
L'opérateur ↑ permet de désigner l'objet pointé.
L'opérateur @ permet de connaitre l'adresse d'un objet adressable. Cette opération permet de faire pointer un pointeur sur une donnée.

La constante nil peut être affectée à un pointeur quelconque, elle permet d'indiquer qu'un pointeur ne repère aucun objet.

Exemple

```
type t = record
         a:integer ;
         b:array  [0..5]  of char ;
         c:boolean ;
         end
var v1,v2 : t ;
        pt1,pt2 : pointer to t ;
        pi      : pointer to integer ;
        pc      : pointer to char ;
        pb      : pointer to boolean ;
        pt1 := nil ;
        pt2 :=   @ v2 ;
        pt1 := pt2 ;
        pi  :=   @ v1.a ; (*  @ (v1.a) *)
        pi  := pt1↑.a ; (* v1.a := v2.a *)
        pc  :=   @ pt1↑.b[5]   ; (*   @ (pt1↑.b[5]) *)
```

remarque 1

L'opération p1 := p2 est autorisée également si p1 est un pointeur quelconque et p2 un pointeur sur un objet de type unspecified.

remarque 2

L'opérateur @ calcule une adresse-mot, si x est un objet non cadré sur une frontière de mot @ x est illégal.

remarque 3

L'opération p := @x où x est de type t est valide si p est un pointeur sur t.

Si x est de type unspecified alors p := @x est autorisé quelque soit le type t pointé par p. Ceci permet de donner à l'exécution un type à une zone mémoire non spécifiée (déclarée unspecified).

remarque 4

Il est possible de faire une référence en avant sur le nom du type de l'objet pointé.

exemple

```
type t1 : record
          a:integer ;
          b:pointer to t2 ;
          end ;
     t2 : record
          a:char;
          d:pointer to t1 ;
          end ;
```

4.4 - Variables

Une déclaration de variable associe un identificateur à une instance d'un type. Les variables dont les identificateurs apparaissent dans une même liste sont de même type.

```
<var-dcl> = <ident-list> ':' <type>  [<initialisation>] .
```

Seules les variables globales peuvent être initialisées. Les variables simples sont initialisées par des constantes de même type. Pour les variables structurées, les initialisations doivent apparaitre entre crochets et les éléments de la structure (éléments d'un tableau, champs d'un enregistrement) sont séparés par des virgules. Pour les tableaux, un facteur de répétition permet d'initialiser plusieurs éléments consécutifs à la même valeur. Le point d'interrogation permet de laisser la valeur d'un élément non précisée. Un tableau de caractères peut être initialisé par une chaîne de caractères.

```
<initialisation> = value <valinit>.
<valinit> = '?' |  <constante> |  '[' <initstruc> ']'.
<initstruc> = <initelem>   {',' <initelem>} .
<initelem> =  [<repetition>]   <valinit>.
<repetition> = '<' <ent-exp> '>'.
```

Exemple :

```
type t1 = record
          a:array  [0..5]  of integer ;
          b:boolean ;
          c:char ;
          end ;
var v1 : t1 value   [[-1,<2>5,?,4,5],true,"A"]  ;
    v2,v3 : array  [1..3]  of char value 'ABC' ;
    v4 : integer value 5 ;
    v5 : boolean ;
```

4.5 - Actions et algorithmes

Un sous-programme (action) est composé de deux parties : une partie déclaration d'action et une partie algorithme.

- La déclaration d'action comprend les spécifications externes du sous-programme : nom et nature du sous-programme ; nom, type et mode de passage de chacun des paramètres.
- L'algorithme réalise l'action ; il contient les objets internes et les instructions du sous-programme.

4.5.1 - actions

```
<action-dcl> = <ident> '=' <action>';'.
<action> = <procedure> | <fonction> | <task>.
<procedure> = [<type resultat>] procedure '(' [<p-param>] ')'.
<fonction> = <type resultat> function '(' [<f-param>] ')'.
<task> = task '(' <f-param> ')'.
<type resultat> = ident | type de base.
```

4.5.1.1 - Nature des actions

Il y a quatre sortes d'action

les procédures Exemple p=procedure (a:integer) ;
les procédures avec résultat
Exemple p= procedure (a:integer) : boolean ;
les fonctions Exemple p= function (a:integer) : boolean ;
les modèles de tâches
Exemple p=task (a:integer) ;

Les procédures avec résultat et les fonctions retournent à l'issue de leur exécution un résultat ; leur appels figurent donc dans une expression. Les fonctions LEGOS sont seulement plus restrictives pour ce qui est du passage des paramètres (voir 4.5.1.3). De plus l'exécution d'une fonction ne doit en aucun cas modifier directement ou indirectement des données non locales à la fonction.

Le type du résultat (d'une fonction ou d'une procédure avec résultat) ne peut être qu'un type de base (ou équivalent) ou bien un pointeur. Ce ne peut en aucun cas être un type structuré.

Les modèles de tâches permettent de créer des actions pouvant s'exécuter concurremment (voir Chapitre 1).

4.5.1.2 - Type des paramètres

Le type d'un paramètré formel peut être :

- un type de base ou le type ref
- le nom d'un type déjà déclaré
- un tableau aux bornes non-spécifiées (array of t)
- un pointeur banalisé (pointer).

Dans le cas d'un paramètre formel a:array of t, dans le corps du sous-programme, le tableau sera considéré comme indicé de 0 à n-1 où n est le nombre d'éléments du tableau effectif passé en paramètre. La fonction high (notée high(a)) permet de connaître la borne supérieure d'un tel tableau.

Dans une liste a,b,c : array of t les paramètres effectifs devront être des tableaux ou des tranches ayant même nombre d'éléments.

Dans le cas d'un paramètre formel a:pointer, le paramètre effectif peut être un pointeur d'un type quelconque.

Dans une liste a,b,c:pointer, les paramètres effectifs devront être de même type.

Les opérations a := c, b = a, a /= b sont alors autorisées ; ainsi que a := @x où x désigne un objet de type unspecified.

La fonction size (notée size(a)) permet de connaitre la taille du type de l'objet pointé par le paramètre effectif correspondant à a.

4.5.1.3 - Mode de passage des paramètres

Il y a quatre modes de passage de paramètres :

in : à l'appel du sous-programme le paramètre formel est initialisé avec la valeur du paramètre effectif (recopie).
C'est le seul mode de passage autorisé (et d'ailleurs implicite) pour les fonctions et les modèles de tâches.

Pour les procédures (avec ou sans résultat), c'est le mode de passage par défaut, mais il existe troix autres modes :

out : le paramètre formel est affecté au paramètre effectif lors du retour au programme appelant.
inout : combine les deux modes précédents.
access : le paramètre formel référence directement le paramètre effectif (passage par référence).

Ces trois derniers modes nécessitent que le paramètre effectif soit un élément adressable (variable, variable pointée, élément de variable..).

```
<p-param> = <p-section>  {';' <p-section>} .
<p-section> = <paramlist> ':' <type param>.
<paramlist> = <param>  {',' <param>} .
<param> =  [<mode>]   ident.
<type param>= [array of](ident | type de base)
<mode> = in | out | inout | access.
<f-param> = <f-section>  {';' <f-section> }.
<f-section> = <identlist> ':' <type param>.
```

4.5.2 - Algorithmes

```
<algorithme> = body ident ';'  {<declaration>} {<algorithme>}
               <instpart> end  [ident] .
<instpart> = begin  [<label>]  <instlist>.
<label> = label <identlist>.
```

Chaque algorithme est introduit par le mot-clé body suivi de l'identificateur de l'action associée.

Un algorithme comprend des objets locaux (déclarations, algorithmes..) ainsi qu'une liste d'instructions (précédée éventuellement d'une liste d'étiquettes). Comme pour les modules, l'identificateur de l'action associée peut figurer après le end terminant l'algorithme. La partie label introduit une liste d'identificateurs qui serviront d'étiquettes de branchement dans le bloc d'instructions associé.

Remarques

L'algorithme, associé à une fonction ne doit en aucun cas modifier des variables non locales, ceci directement ou indirectement. On ne peut donc dans l'algorithme d'une fonction, à l'exception des éventuelles procédures locales, n'appeler que des fonctions.

restrictions :
- Les variables locales d'un algorithme ne peuvent pas être initialisées.
- On ne peut déclarer des actions task dans un algorithme.

5 - EXPRESSIONS

Une expression est une construction permettant de calculer des valeurs à partir d'opérandes (constantes, variables, fonctions, procédures avec résultat) et d'opérateurs.

5.1 - Opérandes

Un opérande (noté <primaire>) peut être constitué par
- une constante littérale
- un identificateur de constante ou de variable
- un appel de fonction ou de procédure avec résultat
  exemple : f(a,b)
- un élément d'une variable structurée (tableau ou enregistrement)
  exemple : v[a] w.b
- une valeur repérée par un pointeur
  exemple : p↑
- une tranche d'un tableau ou d'une variable de type word (c'est-à-dire une suite d'éléments consécutifs).
  exemple : v[2..5] (voir 6.2)
- une expression entre parenthèses.

Par la suite, nous distinguerons et noterons différement
- les appels de fonctions ou de procédures (notation <appel>)
- les constantes littérales (notation <constlit>)
- les tranches (notation <tranche>)
- les autres opérandes étant regroupés sous le nom (abusif) de variable (notation <variable>)

<primaire> = <variable> | <appel> | <constlit> | <tranche> | '('<expression>')'.
<variable> = ident | <variable> '.' ident | <variable> '[' <expression> ']' <variable> ' '.

5.2 - Tranches

Une tranche est une suite d'éléments consécutifs d'un tableau. Si vt est un tableau, la notation vt[i..j] désigne la tranche dont le premier élément est vt[i] et le dernier est vt[j] .

Les bornes d'une tranche peuvent être calculables soit à la compilation soit à l'exécution. Dans le premier cas, on peut, en particulier, si inter a été déclaré comme intervalle, écrire vt[inter].

<tranche> = <variable> '[' <intervalle> ']'.

Une tranche est caractérisée par le nombre et le type de ses éléments. Deux tranches sont dites de même type si leurs éléments sont de même type et en nombres égaux.

Remarques :
- Le type word est assimilé à packed array [o..n] of bit ; si vt est une variable de type word, la tranche vt [i..j] désignera la suite de bits allant de i à j. Pour les tranches de bits, les bornes doivent être calculables dès la compilation.
- Une chaîne de n caractères est assimilée à une tranche de caractères ayant n éléments.

5.3 - Opérateurs

Les opérateurs sont regroupés dans cinq classes selon leur précédence. Par ordre de précédence croissante, les classes d'opérateurs sont les suivantes :

- Opérateurs de relation et d'appartenance
  a) = /= < > <= >= (notation <rel>)
  b) in
- Opérateurs d'addition
  + - or eor orw eorw (notation <add>)
- Opérateurs de multiplication
  * / mod and andw (notation <mul>)
- Opérateurs de décalage
  sll slr scl scr sar (notation <dec>)
- Opérateurs unaires
  + - not notw (notation <un>)

Une séquence d'opérateurs de même précedence est évaluée de gauche à droite

<expression> = <expressionsimple>
[<rel> <expressionsimple> | in <intervalle>].
<expressionsimple> = [<un>] <terme> {<add> <terme>} .
<terme> = <facteur> {<mul><facteur>} .
<facteur> = <primaire> [<dec> [<un>] <primaire>] .

- Opérateurs entiers

Symbole	Opération
+	addition
-	soustraction
*	multiplication
/	division entière
mod	modulo

Ces opérateurs opèrent sur des opérandes de type entier.

- Opérateurs logiques :

Symbole	Opération
or	union
eor	ou exclusif
and	intersection
not	négation

Ces opérateurs opérent sur des opérandes de type booléen.

remarque :
a or b est réalisé par "if a then true else b endif"
a and b est réalisé par "if a then b else false endif"
L'évaluation est arrêtée dès que la valeur de l'expression est connue (évaluation en "short circuit").

- Opérateurs de décalage :

Symbole	Opération
sll	décalage logique à gauche
slr	décalage logique à droite
scl	décalage circulaire à gauche
scr	décalage circulaire à droite
sar	décalage arithmétique à droite

L'opérande de gauche doit être de type word. L'opérande de droite doit être de type entier (il précise de combien de bits on doit décaler l'opérande de gauche)

- Opérateurs logiques-mot

Symbole	Opération
orw	union
eorw	ou exclusif
andw	intersection
notw	négation

Ces opérateurs opèrent sur des opérandes de type word et produisent comme résultat un mot-machine où la valeur de chaque bit est le résultat de l'opération logique effectuée bit à bit sur les deux opérandes.

- Opérateurs de relation

Symbole	Relation
=	égal
/=	différent
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

Ces relations entre opérandes de même type, donnent un résultat booléen. Ces relations s'appliquent aux opérandes de type integer, char, boolean.

Les relations d'égalité-inégalité s'appliquent aussi aux opérandes de type word, pointeur, référence (deux pointeurs (resp. références) sont égaux s'ils désignent la même variable (resp. tâche) ou s'ils sont tous deux à la valeur nil (resp. notask)). Les relations d'égalité-inégalité s'appliquent également aux variables structurées de même type (deux variables structurées sont égales si tous leurs éléments correspondants sont égaux).

Enfin toutes ces relations sont étendues aux tranches de même type, à condition que ces relations soient elles-mêmes définies pour leurs éléments avec le sens suivant :

```
a[i..j]  rel b[k..l]      avec j-i = l-k = n
                          et rel étant un opérateur de relation
l'expression a pour valeur le booléen val ainsi calculé
id := true ; p := 0 ; while (p <= n) and id
                            loop
                            id := a[i+p] = b[k+p] ;
                            val := a[i+p] rel b[k+p] ;
                            p := p+1 ;
                            endloop ;
```

Par exemple, pour les tranches de caractères, ceci permet d'effectuer des comparaisons suivant l'ordre alphabétique.

- Opérateur d'appartenance

Symbole	Opération
in	inclus dans

L'opérateur doit avoir un opérande gauche de type entier et pour opérande droit un intervalle (calculable à la compilation ou à l'exécution). Le résultat est un booléen qui indique si l'opérande gauche est ou non compris dans l'intervalle (bornes incluses).

Remarque :
Pour tous ces opérateurs, les types des opérandes peuvent également être des redéfinitions des types indiqués (voir paragraphe suivant).

6 - EGALITE DE TYPE

Du point de vue de l'égalité des types, un type est caractérisé par son nom. Ce nom peut être :

1) prédéfini : c'est le cas pour integer, boolean, char, word, ref, unspecified
2) précisé par l'utilisateur lors d'une déclaration explicite de type
3) attribué lors de la compilation (nom interne, différent de tous les autres). C'est le cas pour une déclaration de variable structurée lorsque sa structure est explicitement décrite au moment de la déclaration de la variable.

Exemple :

```
var  v1,v2 : array  [1..10]  of integer ;
     v3,v4 : array  [1..10]  of integer ;
```

Ceci est équivalent à :

```
type dummy1 : array  [1..10]  of integer ;
     dummy2 : array  [1..10]  of integer ;
var  v1,v2 : dummy1 ;
     v3,v4 : dummy2 ;
```

Notation :

On notera type(v1), le nom du type de la variable v1 définie ci-dessus. On notera de la même façon le nom du type de l'élément d'une variable structurée (type(v.a), type(v[i])). Enfin si exp est une expression, type(exp) désigne le nom du type du résultat de l'expression (dans ce cas le type est toujours un type prédéfini). Dans une déclaration de type de la forme t = p où p est le nom d'un type prédéfini, on dit que t est un synonyme de p et on note ceci t syn p.

6.1 - Compatibilité des type pour l'assignation et les relations d'ordre

A := B (ou A=B, A<B, ..)

- A étant une variable ou un élément de variable
- B étant une variable, un élément de variable ou une expression.

Cette expression est valide si R1, R2 ou R3 est vérifié.

R1 : type (A) = type (B)
R2 : type (A) syn type (B) ou type (B) syn type (A)
R3 : type (A) syn t et type (B) syn t

6.2 - Compatibilité de type pour le passage de paramêtre

a) mode in

Ce sont les règles précédentes qui s'appliquent

b) mode access, out, inout

Soit pf le paramètre formel et pe le paramètre effectif

- si le type du paramètre formel n'est pas prédéfini, on doit avoir le même nom de type pour pe et pf
  type(pe) = type(pf)
- si le type du paramètre formel est prédéfini il suffit d'avoir
  type(pe) syn type(pf)

7 - INSTRUCTIONS

On distingue les instructions élémentaires (affectation, appel de procédure, branchement) et les instructions composées, qui sont construites à partir d'autres instructions. Chaque instruction peut être précédée d'une (ou plusieurs) étiquette. Cette étiquette doit être déclarée dans la liste d'étiquettes du corps de procédure correspondant. Les instructions sont terminées par des points-virgules.

```
<inst-list> = <inst> { <inst>}.
<inst> = [<ident-list> ':'] <instruction>';'.
<instruction> = <affectation> | <appel> | <goto> | <if> |
                <case> | <loop> | <for> | <until> |
                <while> | <create> | ϕ.
```

7.1 - Affectation

Cette instruction permet de modifier la valeur d'une variable ou d'une tranche. Une variable de type enregistrement peut être modifiée champ par champ par un ensemble (entre crochets) d'expressions spécifiant les valeurs des champs successifs de l'enregistrement. Si la valeur n'est pas spécifiée (champ '?'), le champ correspondant n'est pas modifié.

L'instruction d'affectation s'applique aux tranches de même type (y compris les tranches de bits).

```
<affectation> = <affectvar> | <affecttranche>.
<affectvar> = <variable> ':=' (<expression> | <multiple>).
```

```
<multiple> = '[' <elem> {',' <elem>} ']'.
<elem> = <expression> | '?'.
<affecttranche> = <tranche> ':=' (<tranche> | chaîne | hexa).
```

Exemple 1 :

```
var a,b:integer ;
    c,d:array[1..10] of integer ;
    e,f:record
        g,h:integer ;
        i,j:array [1..5] of char ;
        end ;
    g:word ;
a := b+5 ;
c := d ;
e := f ;
c[5] := a+d[3] ;
e.i := e.j ;
e.j := 'ABCDE' ;
f := [+3, b+1, ?, 'IJKLM'] ;
```

Exemple 2 :

```
var a,b:word ;
    c:array [1..10] of char ;
    d,e:array [5..15] of integer ;
    a[0..7] := #E3 ;
    c[7..9] := 'ABC' ;
    c[1..3] := c[7..9] ;
    d[5..8] := e[7..10] ;
```

Remarques :

- Une constante hexadécimale peut être affectée à une variable word ou à une tranche de bits. Dans ce dernier cas, il s'agit d'une affectation avec troncation sur les bits de fort poids.

  exemples :

```
a:= #F4BC ;
a[0..3] := #C8     (* a=#84BC *)
a[8..15] := #F     (* a=#840F *)
```

- Une constante chaîne de longueur n peut être affectée à un tableau de n caractères ou à une tranche de caractères de longueur n.

7.2 - Appel de sous-programme

L'instruction d'appel de sous-programme provoque l'exécution du sous-programme indiqué. L'instructon d'appel précise la valeur des différents paramètres. Les paramètres d'appel (paramètres effectifs) doivent être de types compatibles avec les paramètres correspondants de la déclaration (paramètres formels) selon les règles définies au paragraphe 6.2. L'appel de fonction ou de procédure avec résultat suit les mêmes règles. Il n'apparaît pas comme instruction mais comme opérande dans une expression.

Pour les passages en mode out, inout, access, le paramètre effectif peut être une variable, un élément de variable structurée, une valeur pointée, ou une tranche.

Pour un passage de mode in, le paramètre effectif peut aussi être une expression de même type que le paramètre formel.

Passage de paramètre tableau-variable

Lorsque le type d'un paramètre formel est de la forme array of t le paramètre effectif doit être un tableau ou une tranche dont les éléments sont de type t.

Si la déclaration formelle est de la forme : a,b : array of t, alors les paramètres effectifs correspondant à a et b doivent être compatibles (même type d'éléments et même nombre d'éléments).

Passage de paramètre pointeur détypé :

Lorsque le type d'un paramètre formel est de la forme pointer, le paramètre effectif doit être un pointeur.

Si la déclaration formelle est de la forme : a,b : pointer, alors a et b doivent être des pointeurs de même type (c'est-à-dire, pointant des objets de même type).

7.3 Branchement

<goto> = goto ident.

L'identificateur suivant le goto doit avoir été déclaré dans la partie étiquette du bloc courant. L'exécution continue à l'instruction étiquettée par le même identificateur.

L'instruction de branchement et l'instruction étiquettée doivent faire partie d'un même bloc d'instructions. On ne peut donc se brancher à une instruction externe au bloc courant.

Si, dans une instruction composée, une liste d'instructions comporte des étiquettes, ces étiquettes ne peuvent être référencées que depuis l'intérieur de cette liste. La séquence suivante est par conséquent interdite.

```
goto etiq ;
if a=3 then etiq : a := a+1 endif ;
```

7.4 - Instructions avec sélecteur

Ces instructions permettent, à l'exécution, de choisir une liste d'instructions parmi plusieurs selon la valeur d'un sélecteur. Ce sélecteur est de type booléen (instruction if) ou entier (instruction case).

7.4.1 - Instruction IF

<if> = if <expression> then <instlist>
[else <instlist>] endif.

Si la valeur de l'expression booléenne est vraie, alors on exécute la première liste d'instructions, sinon on exécute la seconde liste d'instructions (si elle existe). Le contrôle est ensuite passé à l'instruction qui suit endif.

7.4.2 - Instruction CASE

<case>= case <expression><altern> {<altern>}
[<others>] endcase.
<altern> = when <cas> { ',' <cas>} ':' <inst-list>.
<cas> = <ent-exp> | <intervalle>.
<others> = others <inst-list>.

Une instruction case comporte :
- Un sélecteur entier dont la valeur est calculée à l'exécution
- Une série de liste d'instructions, chacune d'elles étant précédée d'une liste de cas, et éventuellement une alternative others.

Les cas sont des expressions entières ou des intervalles, calculables dès la compilation. Les intervalles ne doivent pas se chevaucher. Toutes les expressions doivent avoir des valeurs différentes et n'appartenir à aucun des intervalles.

A l'exécution,
- On calcule la valeur sel du sélecteur
- S'il existe un cas c tel que l'on ait sel = c ou sel in c, on exécute la liste d'instructions associée à c puis le contrôle est passé derrière le endcase
- sinon, soit l'alternative others est exécutée quand elle existe, soit une erreur est signalée.

Exemple :

```
case i
   when -5,0..3 : y := 20;          (* exécuté si i=-5,0,1,2,3 *)
   when -3..-1,5..8 : y := y+4;     (* exécuté si i=-3,-2,-1,5,6,7,8 *)
   others y := 0;                   (* exécuté si i<-5, i>8 ou i=-4,4 *)
endcase
```

7.5 - Instructions répétives

Ces instructions permettent d'exécuter plusieurs fois (éventuellement aucune fois) une même liste d'instruction.

Instruction LOOP

<loop> = loop <instlist> endloop.

La liste d'instructions entre loop et endloop est exécutée indéfiniment.

Instruction WHILE

<while> = while <expression> <loop>.

L'expression booléenne est évaluée à l'entrée de l'instruction while et après chaque tour de boucle. Tant que l'expression est vraie, la boucle est répétée sinon l'instruction while est terminée.

Instruction UNTIL

<until> = <loop> until <expression>.

L'expression booléenne est évaluée après chaque boucle. La répétition s'arrête dès que la valeur de l'expression est vraie. Sinon on répète la boucle. La boucle est exécutée au moins une fois.

Instruction FOR

<for> = for ident (up | down) <intervalle> <loop>.

L'identificateur suivant le mot clé for, appelé indice de la boucle, est considéré comme une déclaration locale implicite à l'instruction for. L'indice de la boucle est considéré comme une constante entière non modifiable, ni directement par affectation, ni indirectement (paramètre acces , out, inout d'un sous-programme). Cet identificateur, local, n'est pas référençable hors de la boucle. Les bornes de l'intervalle que parcourra l'indice de boucle sont évaluées une fois pour toutes au début de l'instruction.

L'intervalle sera parcouru dans le sens croissant (resp. décroissant) si on utilise up (resp. down).

Exemple :

```
while succ loop a[i].val := x ;
              succ := a[i].fils ;
              endloop;
loop a[i] := i+1 ; i := i+1 ; endloop until i=2000 ;
for j up 1..100 loop a[j] := j endloop ;
for j down b[i].. b[i] + 10 loop...endloop ;
```

7.6 - Instruction de création de tâches

<create> = create <variable> from <appel>.

Cette instruction permet d'instancier une tâche à partir du modèle de tâche précisé dans la partie <appel>.

La partie <appel> se décrit comme un appel de sous-programme où l'action est de la forme task et où les paramètres précisent les valeurs d'initialisation de la tâche.

La partie <variable> décrit un objet de type ref qui servira de repère de tâche.

Cette instruction ne lance pas la tâche. Le lancement se fera à l'aide d'une procédure décrite dans un noyau système (standard ou propre à l'utilisateur).

8 - INTERFACES

Une interface permet d'encapsuler un ou plusieurs modules, en ne laissant visibles de l'extérieur que certains objets. Ces objets peuvent être des constantes, des types, des variables et des en-têtes de sous-programmes. L'interface est construite a posteriori à partir de modules déjà compilés dont les noms apparaissent derrière le mot interface.

Les types ainsi exportés peuvent l'être soit avec leur structure visible (elle doit alors être répétée dans l'interface) soit avec leur structure cachée (attribut hidden).

Les variables exportées peuvent l'être en étant protégées contre toute modification et ne peuvent alors qu'être consultées (attribut protected).

Il est aussi possible de renommer les objets exportés.

L'utilisation des interfaces est expliquée dans les mécanismes de programmation modulaire au chapitre 1.

```
<interface> = <inter-entête> {<inter-déclaration>} end [ident].
<inter-entête> = module ident interface <ident-list>
                 [use <ident-list>] ';'.
<inter-déclaration> = const {<inter-const-dcl>}|
                      range {<inter-range-dcl>}|
                      type {<inter-type-dcl>}|
                      var {<inter-var-dcl>}|
                      action {<inter-action-dcl>}|.

<inter-const-dcl> = <inter-nom> '=' <constante>.
<inter-range-dcl> = <inter-nom> '=' <intervalle>.
<inter-type-dcl> = <inter-nom> '=' (hidden|<type>).
<inter-var-dcl> = <inter-varlist> ':' (identitype de base).
<inter-action-dcl> = <inter-nom> '=' <action>.

<inter-var-list = <inter-var> {',' <inter-var>}.
<inter-var> = [protected] <inter-nom>.
<inter-nom> = [ident is] ident.
```

Exemple :

```
module fichspec interface fichier ;
    type fich is file = hidden ;
    action ouvrir is open = procedure (inout f : fich) ;
           fermer is close = procedure (inout f : fich) ;
end fichspec
```

ANNEXE: Exemple d'un programme écrit en LEGOS

L'exemple suivant décrit un allocateur de mémoire.

L'allocation se fait dans une zone de mémoire (heap) dont les tranches libres sont repérées en mémorisant dans deux tableaux parallèles (first et last) la première et la dernière cellules libres d'une zone.

Cet exemple comprend 2 modules l'un définissant les procédures d'allocation-désallocation, l'autre décrivant une réalisation possible.

Module de spécification:

```
module allocator; (* spécification *)

action

allocate = procedure(inout p:pointer; size:integer);
(* allocate alloue une zone de mots machine de longueur égale à size *)
(* le premier mot de cette zone sera repéré par p  *)

deallocate =procedure(inout p:pointer; size:integer);
(* deallocate restitue une zone de mots machine de longueur égale à size *)
(* à partir de l'adresse pointée par p *)

end allocator
```

Module de réalisation:

```
module allocreal use allocator,errormodule;(* réalisation *)

const m=100;
      n=1000;

var   first,last:array [ 1..m ] of integer ;
      current : integer;
      heap    : array [ 1..n ] of unspecified;

body allocate;
        var i:integer;
        begin
        i:=1;
        (* recherche d'une zone libre suffisamment grande *)
        while (i<=current) and (size>last[i]-first[i]+1)
                  loop inc(i); endloop;
        if i<=current then (* il existe une zone qui convient *)
                           p:= @heap[i];
                           inc(first[i],size);
                      else erreur(heapfull); endif;
        end; (* of  allocate *)
```

```
body deallocate;
        var i,j:integer;
        begin
        i:=1;
        j:=p-@heap; (* calcul de l'index dans le tas *)
        (* recherche de l'endroit où insérer cette nouvelle zone libre *)
        while j>first[i] loop inc(i); endloop;
        if j=last[i-1] then
                        (* insertion en queue *)
                        inc(last[i-1],j);
        else
        if j+size = first[i] then
                             (* insertion en tête *)
                             first[i]:=j;
        else (* création d'une nouvelle zone libre *)
        if current<m then
                         for k downto current .. i
                          loop
                          first[k+1]:=first[k];
                          last[k+1]:=last[k];
                          endloop;
                         first[i]:=j;
                         last[i]:=j+size-1;
        else erreur(heapfull); endif;
        endif;
        endif;
        p:=nil;
        end; (* of deallocate *)


(* initialisation *)
begin
current:=1;
first[current]:=1;
last[current]:=1000;
end allocreal
```

remarques:

- inc est une procédure prédéfinie réalisant l'incrémentation.
  inc(x,i) = x+i
- errormodule est un module où est défini le traitement des erreurs.

Les fonctions Pascal new et dispose apparaissent alors comme des macros réalisées de la façon suivante:

ayant

```
type t=... ;
var  p: pointer to t;
```

- new(p) se traduit par allocate(p,size(t)).
- dispose(p) se traduit par deallocate(p,size(t)).

où size(t), fonction prédéfinie, donne la taille en mots-machine d'un objet de type t.

BIBLIOGRAPHIE

Rapports

ICHBIAH J., HELIARD J.C., RISSEN J.P., COUSOT P. [1974]
The System Implementation Language LIS - reference manual
[CII - Technical Report 4549E/EN]

COUHAULT A.M., COUSOT P., ICHBIAH J., TERRINE G. [1976]
Introduction très informelle au langage LIS
[Rapport IRIA-SESORI, projet SFER]

LUCAS M. [1977]
Projet Sésame. Conception modulaire des systèmes d'exploitation. Outils pour la programmation modulaire.
[USM Grenoble - Thèse de 3° Cycle]

MITCHELL J.G., MAYBURY W., SWEET R. [1978]
Mesa Language Manual
[Xerox Parc CSL-78-1]

WIRTH N. [1977]
Modula : A Language for Modular Multiprogramming
[Practice and Experience, 7, 3-35]

WIRTH N. [1978]
Modula 2
[Tech. Report 27, Institut fur Informatik ETH - Zurich]

Articles

BOUCHENEZ J.L., LOYER M., PRUSKER F., SOGNO J.C. [1978]
LEGOS : Langage d'écriture de logiciels sur mini et micro-ordinateurs
[Revue BIGRE n° 12]

BOUCHENEZ J.L., LOYER M., LUCRECE L., MAURICE P., PRUSKER F., SOGNO J.C. VERCOUSTRE A.M. [1980]
Le Système LEGOS - Environnement de programmation sur Mitra 225
[Journées BIGRE, Rennes Décembre 1980]

LOYER M. [1981]
Les mécanismes de programmation modulaire dans le langage LEGOS
[Journées Francophones, Genève Janvier 1981]

WIRTH N. [1979]
The Module : A system structuring facility in high level programming languages
[Proceedings of the Symposium on Language design and Programming methodology, Sydney 10-11 September 1979 - Springer Verlag]

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique